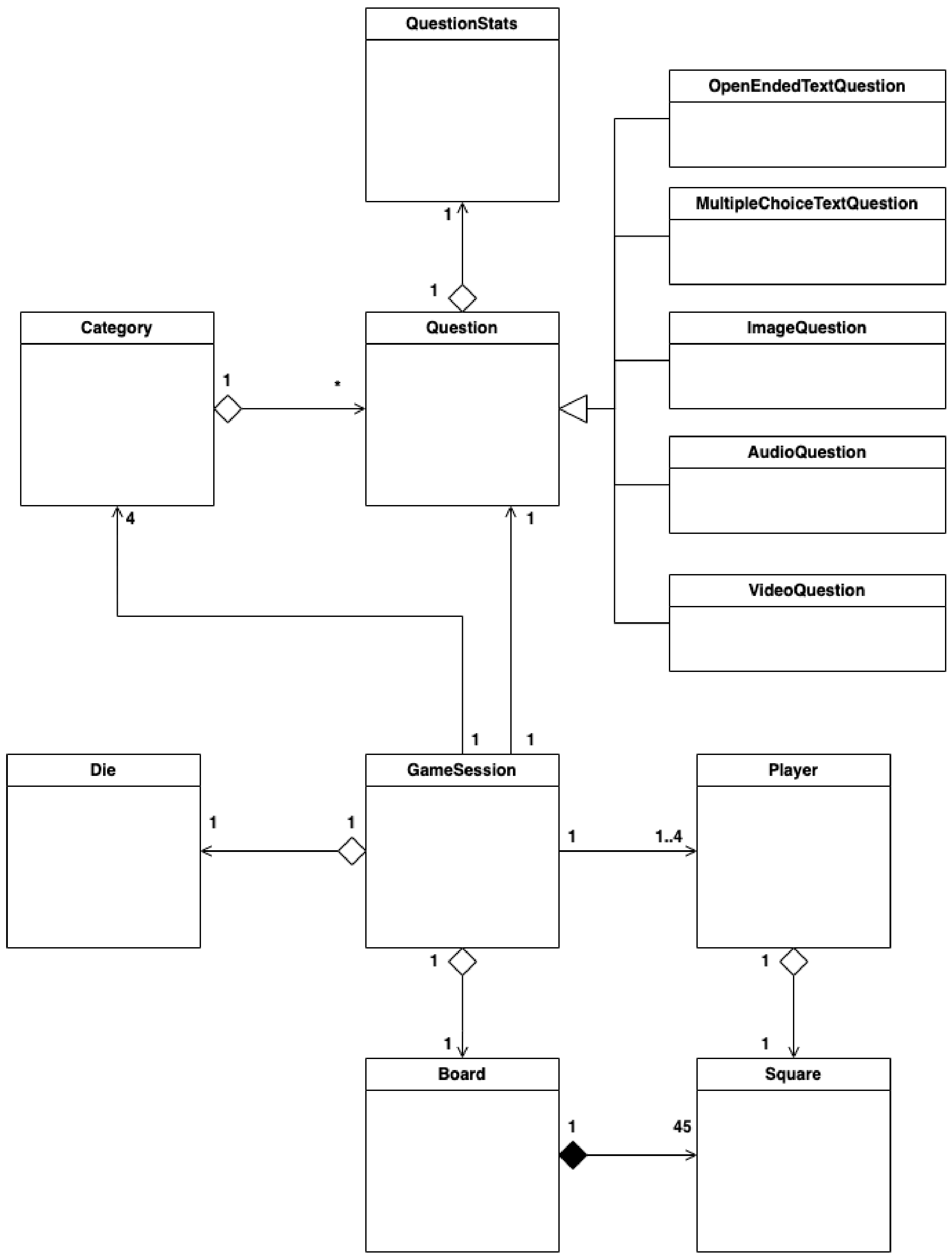
Design Document

Caffeine Coders | 30 July 2024

Fueling innovation, one cup at a time!

Document Table of Contents:

Static Design



Class Specifications

GameSession

The GameSession class holds crucial information about the ongoing game, such as categories, players, the game board, the die, the current player, and the current question. It also manages the sequence of turns, ensuring that each player gets their turn in the correct order and encapsulates the actions that can occur during a player's turn

Attributes:

- **categories: dict(Color, Category) -** Stores the game categories associated with different colors.
- **players: list(Player) -** Keeps track of all players in the game session.
- **gameboard: Board** - Represents the game board.
- **die: Die** - Represents the die used in the game.
- **currentPlayer: Player** - Identifies the player whose turn is currently active.
- **currentPlayerIndex: int** - Index to track the position of the current player in the players' list.
- **currentQuestion: Question** - Holds the question being asked currently.

Methods:

- **getBoard(): Board -** Retrieve this session's game board.
- **startTurn(): int -** Initiates the start of a player's turn.
- **endTurn() -** Ends the current player's turn and prepares for the next player's turn.
- **changeTurn(newPlayerIndex: int) -** Changes the turn to a specified player index.
- **rollDie(): -** Simulates rolling the die.
- **getAvailableDirections(): list[Direction] -** This method retrieves the directions available to the current player from their current position.
- **pickDirection(direction: Direction) -** Allows the current player to pick a direction to move to the next square position.**getQuestion():** Get a question from the category that corresponds to the color of the square that the current player is on.
- **selectCategory(color: Color): Category -** Allows a player to select a category based on a color.
- **getColors(): list[Color] -** Get a list of available category colors.
- **evaluateAnswer(answer: String): Bool** - Evaluates if the given answer is correct.
- **awardPoints(playerIndex: int): Bool -** Awards points to a player based on their index.
- **checkForWinner(): Bool**- Checks if the current player has won the game.
- **endGame(): dict -** Ends the game and returns the final game state.

Category

The Category class in the image is responsible for organizing and managing the different categories of questions within the game

Attributes:

- **name: String -** Holds the name of the category
- **questions: list(Question) -** list that contains all the questions that belong to this category

Methods:

- **pickRandomQuestion(): Question -** Selects and returns a random question from the list of questions in the category

Player

Designed to represent a player within the game, managing their attributes, actions, and interactions.

Attributes:

- **name: String -** The name of the player.
- **grade: String -** The grade level of the player.
- **email: String -** The email address of the player.
- **tokenColor: Color -** The color of the player's token, which is used to visually identify them on the game board.
- **score: dict(Color, boolean) -** This dictionary holds the player's score, keyed by category, with a boolean indicating whether the player has answered an HQ question from that category correctly
- **position: int -** The player's current position on the game board
- **square: Square -** The current square on which the player is located.

Methods:

- **movePlayer(direction: Direction) -** This method moves the player to the next space. As a precondition, this direction must be a valid direction to move the player.
- **setPosition(position: int) -** This method sets the player's current position.
- **setSquare(square: Square)** - This method sets the player's current square.
- **getPosition(): int -** This method returns the current position on which the player is located.

- **getSquare(): Square -** This method returns the current square on which the player is located.
- **updatePoints(color: Color) -** This method is used when a player answers an HQ question correctly, to award the player a point based on the color of the question.
- **isWinner(): Bool -** This method checks if the player has met the winning conditions of having all four categories answered

Board

Board class represents the game board where the game session takes place. It handles the layout of the squares, their positions, and the overall structure of the board.

Attributes:

- **squares: dict(int, Square) -** This dictionary holds all the squares on the board, with each square identified by its position (an integer between 1-45).
- **centerSquare: Square -** This attribute represents the central square of the board.
- **hqSquares: dict(Color, Square) -** This dictionary holds headquarters squares, categorized by color.

Methods:

- **getSquare(position: int): Square -** This method returns the square at the specified position
- **getNeighbors(position: int): list[Direction]** - This method returns the valid directions you can move from the given position.

Square

The Square class represents individual squares on the game board. Each square has specific attributes and methods that define its properties and behaviors.

Attributes:

- **color: Color -** Represents the color of the square.
- **multipleDirector: Bool -** A boolean indicating if the square is a multi-director square
- **position: int -** The position of the square on the board
- **isHQ: boolean -** boolean indicating if the square is a headquarters (HQ) square.
- **isCenter: boolean -** A boolean indicating if the square is the center square on the board.

- **isRollAgain: boolean -** A boolean indicating if landing on this square allows the player to roll the dice again**.**
- **neighbors: map[Direction, int] -** A list of neighboring positions. This represents the squares adjacent to the current square.

Methods:

- **getColor(): Color -** Retrieve this square's color.
- **getIsHQ(): boolean** - Retrieve whether this square is an HQ square.
- **getIsCenter(): boolean** - Retrieve whether this square is the center square.
- **getIsRollAgain(): boolean** - Retrieve whether this square is a roll-again square.

Die

The Die class represents a die used in the game, providing attributes and methods to simulate rolling and handling the die's properties.

Attributes:

- **numberOfSides: int -** Represents the number of sides on the die. This is an integer value indicating how many faces the die has (e.g., 6 for a standard die).
- **currentValue: int -** The current value shown on the die after a roll. This represents the result of the most recent roll.

Methods:

- **roll(): int -** Simulates rolling the die and returns the resulting value. This method generates a random number between 1 and the number of sides on the die.
- **getValue(): int -** Returns the current value of the die
- **setNumberOfSides(sides: int): void -** Sets the number of sides on the die
- **getNumberOfSides(): int -** Returns the number of sides on the die.

Question

The Question class represents a generic question in the game, providing attributes and methods to manage different types of questions and their properties.

Attributes:

- **questionTitle: String -** Represents the title or text of the question.

- **typeOfQuestion: QuestionType -** Specifies the type of the question (e.g., Free Text, Multiple Choice, Audio, Video). This uses the QuestionType enumeration to indicate the type.
- **questionStats: QuestionStats -** An instance of the QuestionStats class, which tracks statistics related to the question, such as how many times it has been asked and answered correctly.
- **dateCreated: String -** The date when the question was created.
- **difficultyLevel: String -** Represents the difficulty level of the question (e.g., Easy, Medium, Hard)
- **creator: String -** The name or identifier of the person that created the question
- **answer: String -** The correct answer to the question.

Methods:

- **getAnswer(): String -** A method to display or return the correct answer

QuestionStats

The QuestionStats class provides functionality to track and update statistics related to questions in the game.

Attributes:

- **timesAsked: int -** This attribute keeps track of how many times the question has been asked.
- **timesCorrectlyAnswered: int -** This attribute keeps track of how many times the question has been answered correctly by the players.

Methods:

- **updateStats(correct: bool): void -** This method updates the statistics of the question based on whether the player's answer was correct or not.
- **calculatePercentage(): float -** This method calculates and returns the percentage of correct answers for the question.

OpenEndedTextQuestion

The OpenEndedTextQuestion class represents a specific type of question where players provide open-ended text responses. This class inherits from the Question class.

Attributes:

- **question: String** - This attribute holds the text of the open-ended question that will be presented to the players.

MultipleChoiceTextQuestion

The MultipleChoiceQuestion class represents a specific type of question where players select from a set of predefined options. This class inherits from the Question class.

Attributes:

- **question: String** - This attribute holds the text of the multiple-choice question that will be presented to the players.
- **options: Array of Strings -** This attribute holds the list of possible answers from which players can choose. Each element in the array represents a different choice.

ImageQuestion

The ImageQuestion class represents a specific type of question where the question or part of the question is provided in an image format. This class inherits from the Question class.

Attributes:

- **imageURL: String -** This attribute stores the path or URL to the image file associated with the question. This file contains the image content that players need to view in order to answer the question.

AudioQuestion

The AudioQuestion class represents a specific type of question where the question or part of the question is provided in an audio format. This class inherits from the Question class.

Attributes:

- **audioURL: String -** This attribute stores the path or URL to the audio file associated with the question. This file contains the audio content that players need to listen to in order to answer the question.

VideoQuestion

The **VideoQuestion** class represents a specific type of question where the question or part of the question is provided in a video format. This class inherits from the Question class.

Attributes:

- **videoURL: String -** This attribute stores the path or URL to the video file associated with the question. This file contains the video content that players need to watch in order to answer the question.

Enumerations

Color

- **Values:**
    - RED = "red"
    - GREEN = "green"
    - BLUE = "blue"
    - YELLOW = "yellow"
    - WHITE = "white"

QuestionType

- **Values:**
    - FREE_TEXT = "free_text"
    - MULTIPLE_CHOICE = "multiple_choice"
    - AUDIO = "audio"
    - VIDEO = "video"
    - IMAGE = "image"

Direction

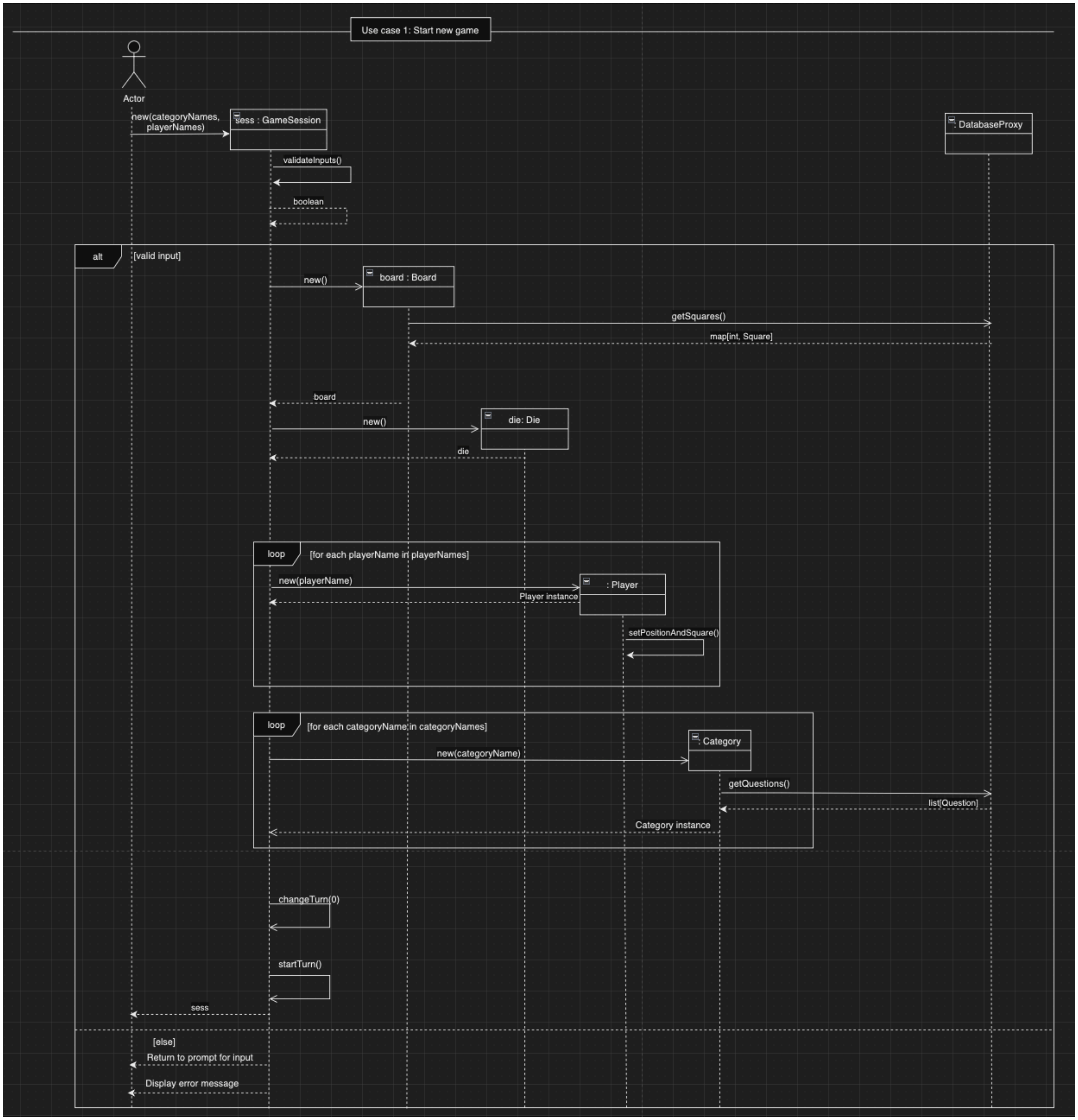
- **Values:**
    - UP = "up"
    - DOWN = "down"
    - LEFT = "left"
    - RIGHT = "right"

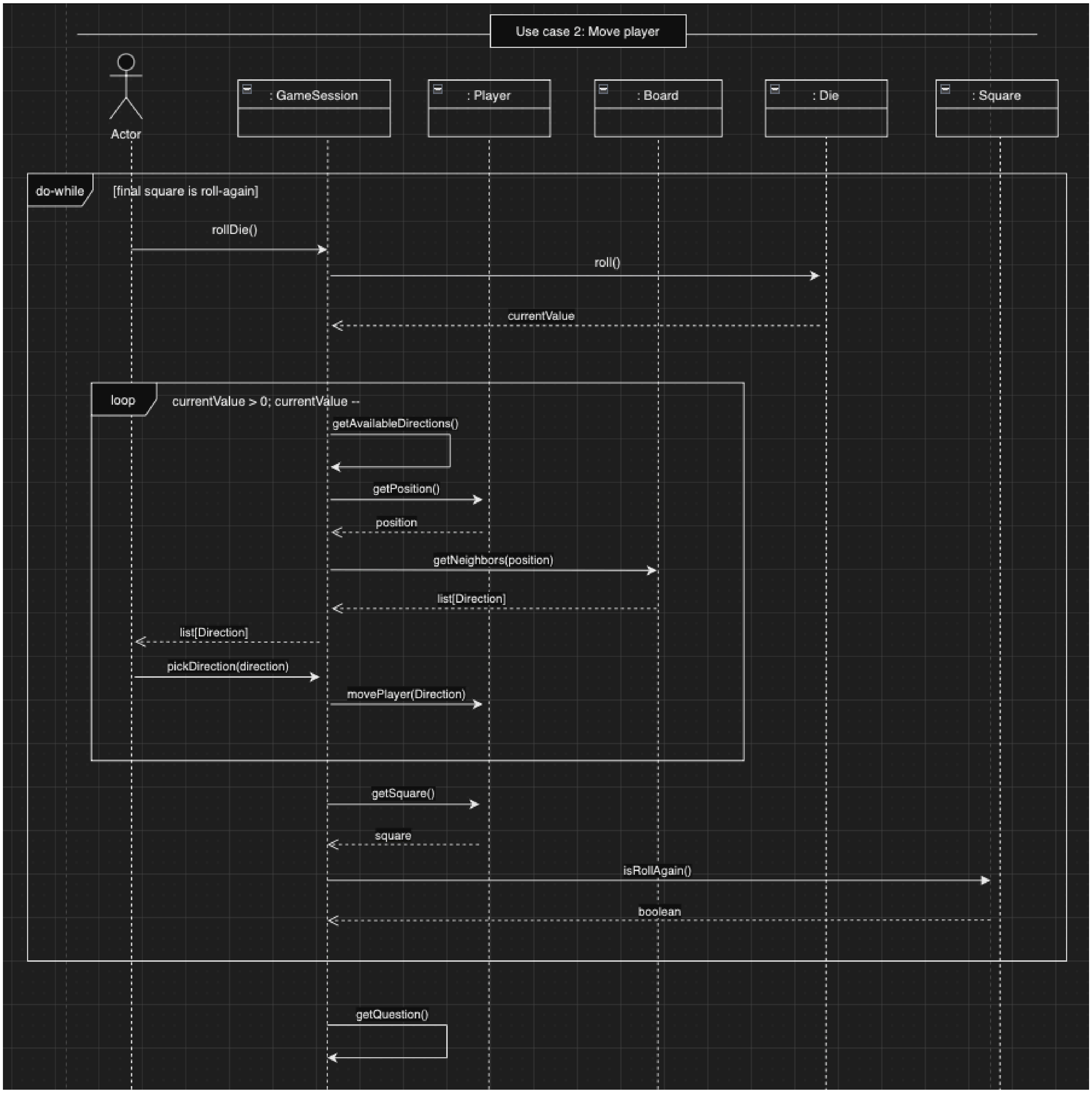
Associations and Compositions

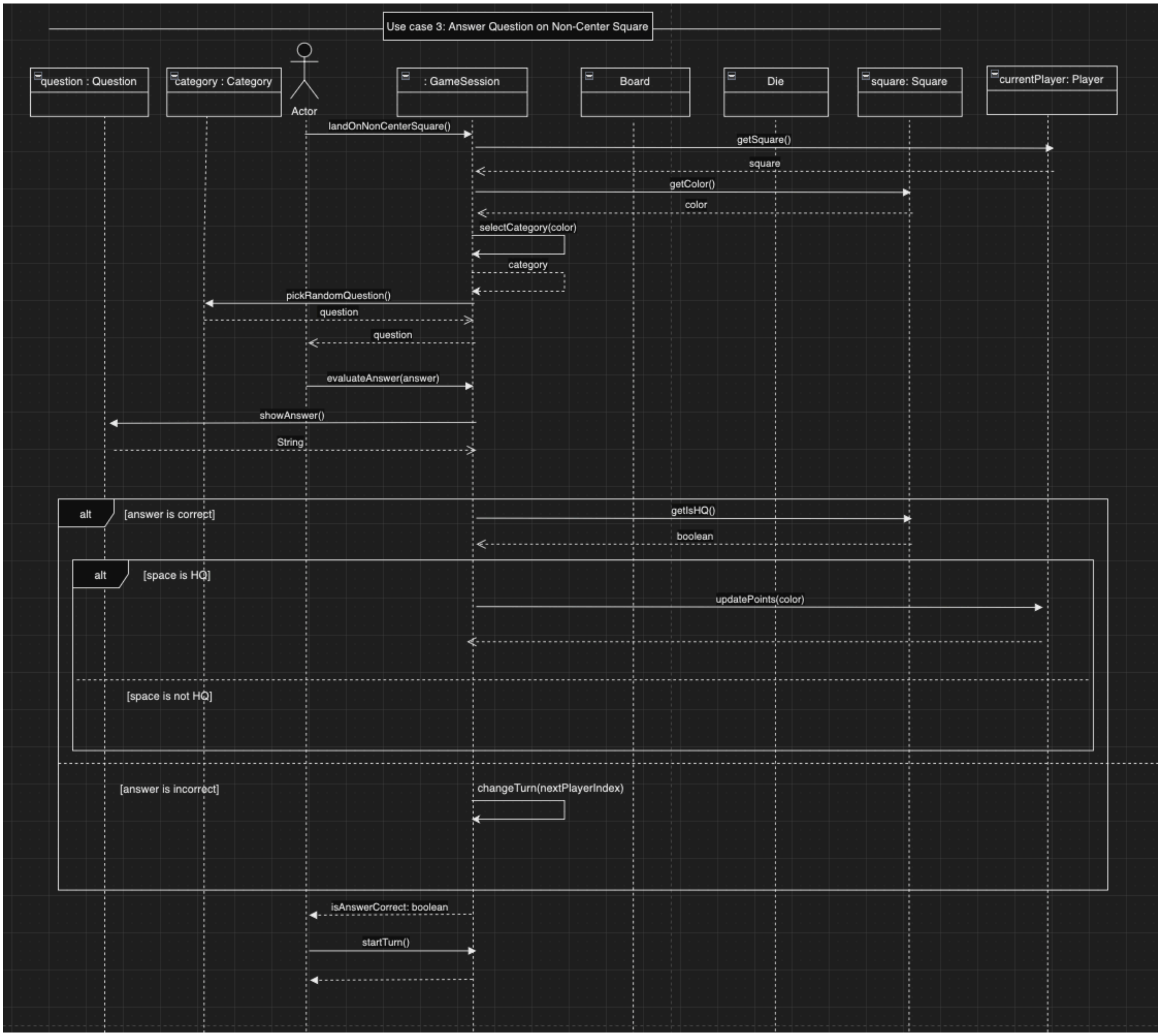
The diagram shows various associations and compositions:

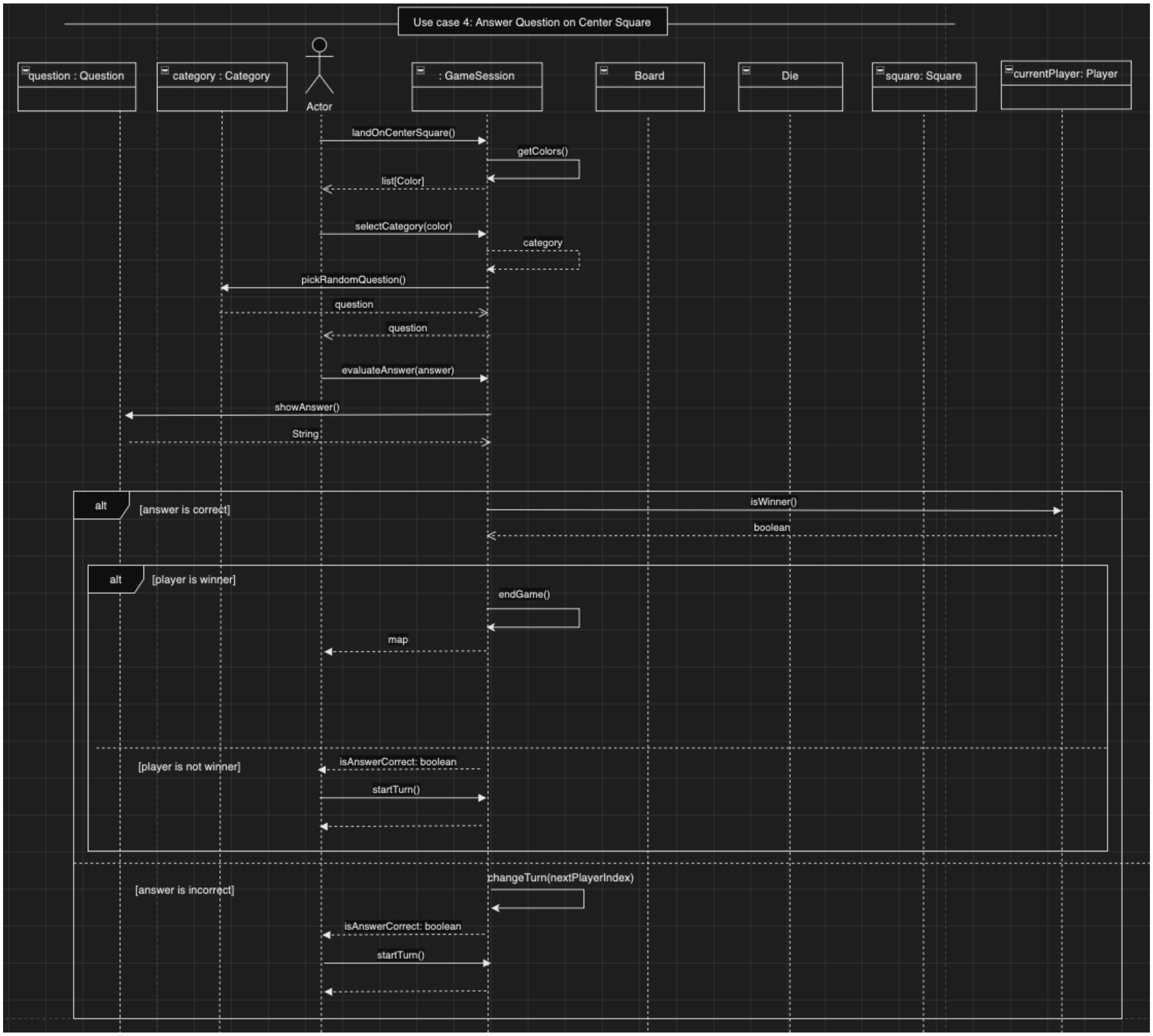
- **GameSession** is associated with **Category, Question, Player.** It is an aggregate of **Board, Die.**
- **Category** has a list of **Question**.
- **Player** is associated with **Square**.
- **Board** is composed of multiple **Square** instances.
- **Question** is associated with **QuestionStats**.
- The specialized question types inherit from **Question.**

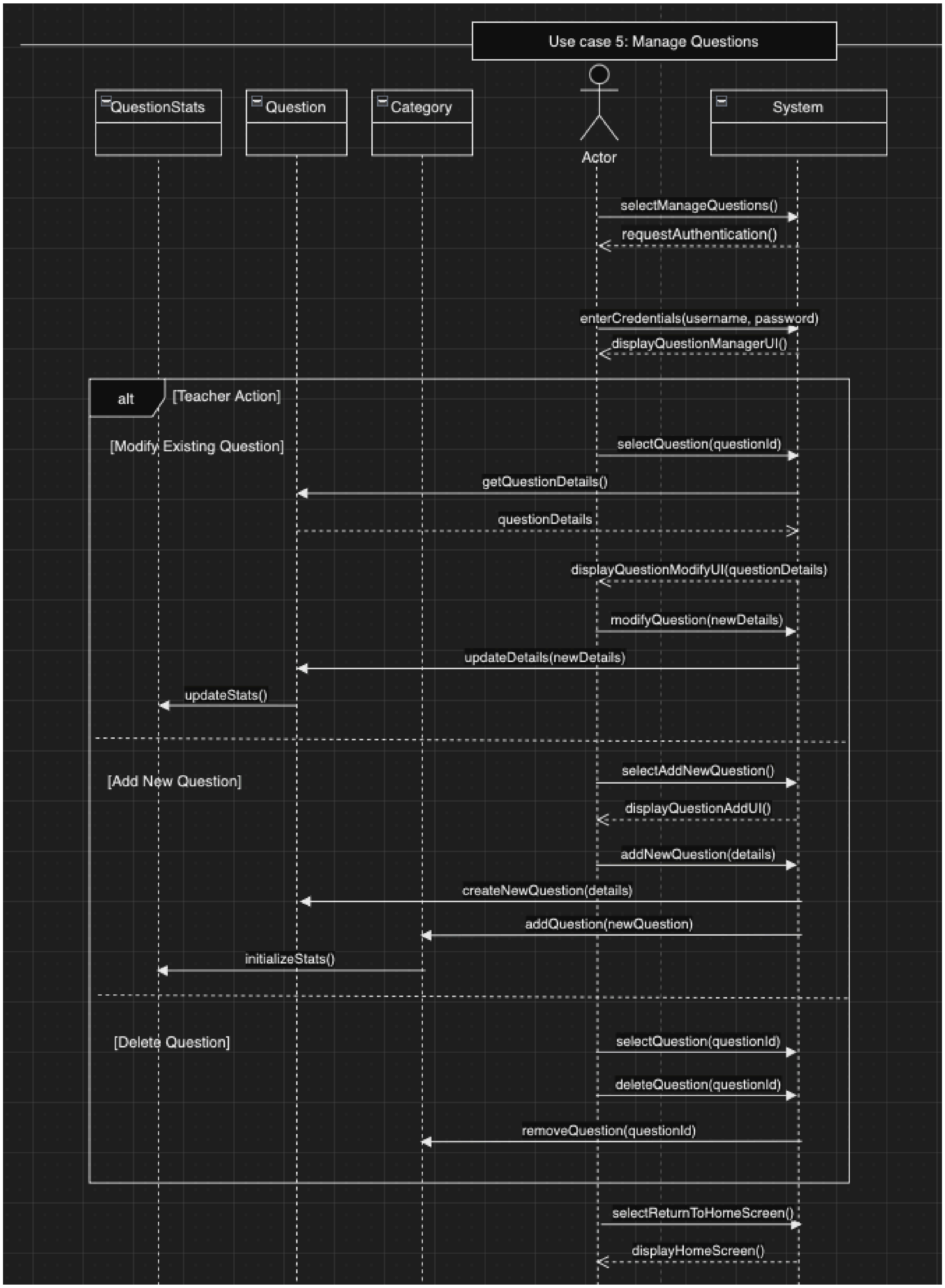
Dynamic Design



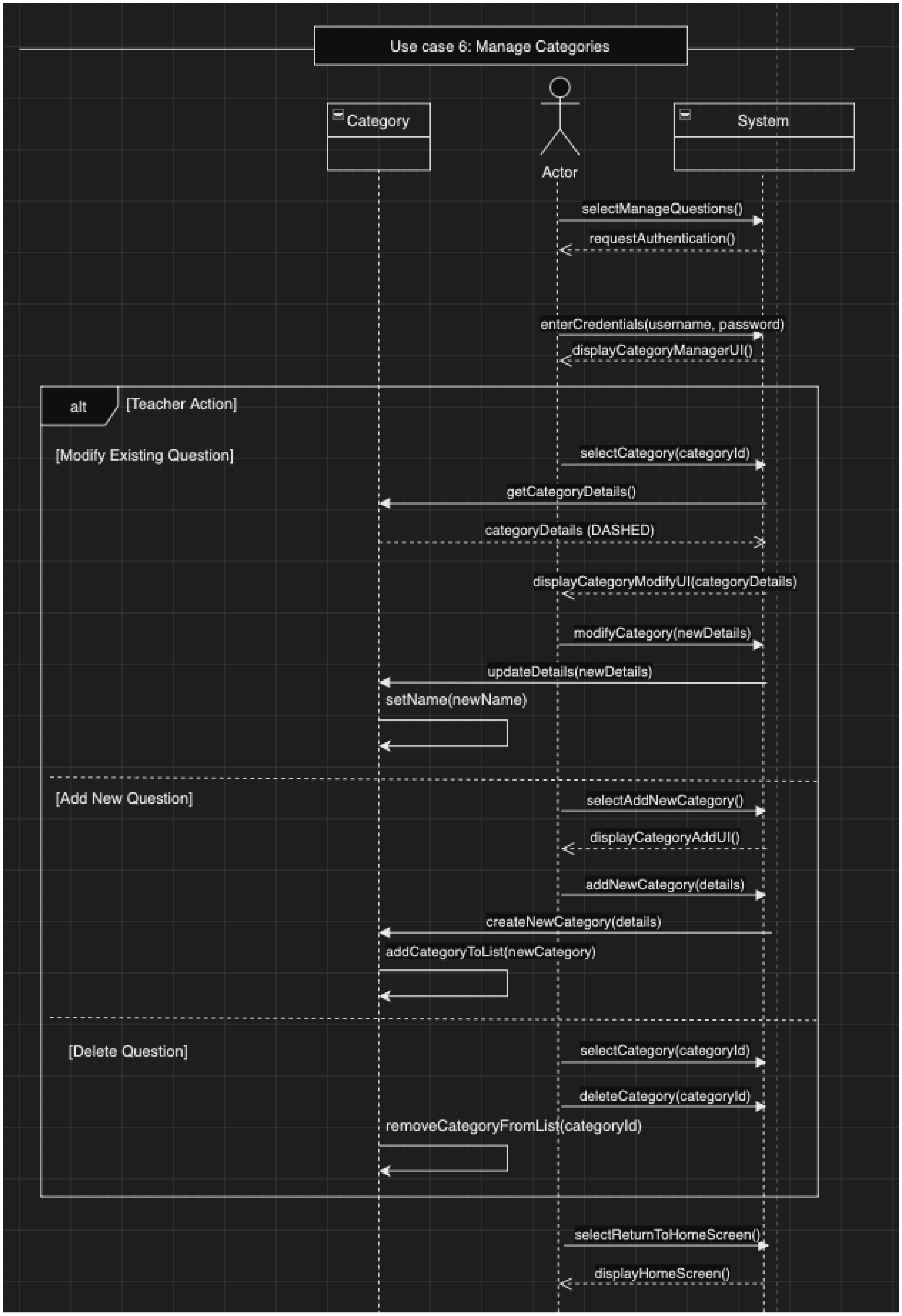

Use case 2: Move player
Actor
: GameSession
: Player
: Board
: Die
: Square
do-while
[final square is roll-again]
rollDie()
roll()
currentValue
loop
currentValue > 0; currentValue --
getAvailableDirections()
getPosition()
position
getNeighbors(position)
list[Direction]
list[Direction]
pickDirection(direction)
movePlayer(Direction)
getSquare()
square
isRollAgain()
boolean
getQuestion()

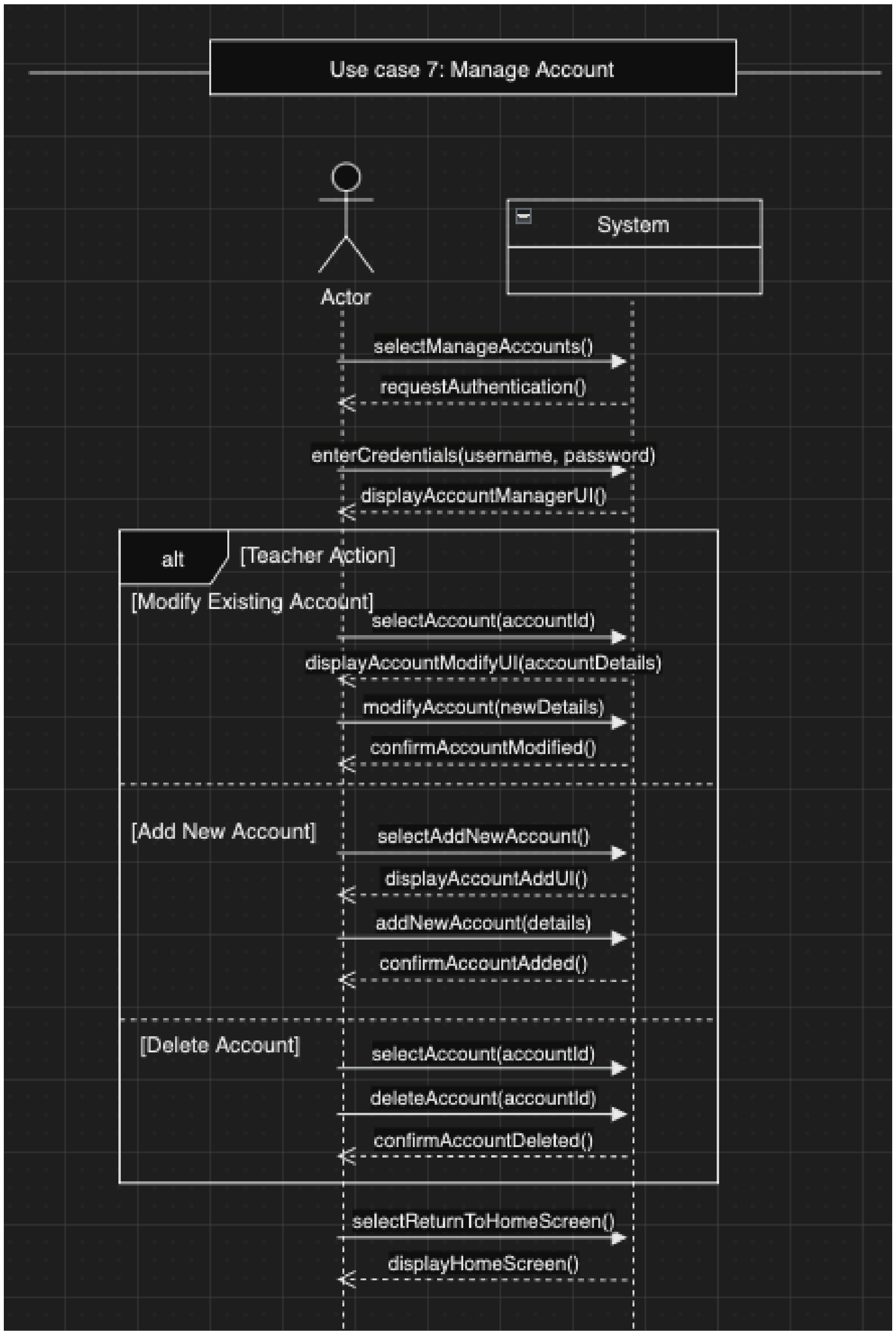

Use case 3: Answer Question on Non-Center Square
question : Question
category : Category
Actor
: GameSession
Board
Die
square: Square
currentPlayer: Player
landOnNonCenterSquare()
getSquare()
square
getColor()
color
selectCategory(color)
category
pickRandomQuestion()
question
question
evaluateAnswer(answer)
showAnswer()
String
alt
[answer is correct]
getIsHQ()
boolean
alt
[space is HQ]
updatePoints(color)
[space is not HQ]
[answer is incorrect]
changeTurn(nextPlayerIndex)
isAnswerCorrect: boolean
startTurn()


Use case 4: Answer Question on Center Square
question : Question
category : Category
Actor
: GameSession
Board
Die
square: Square
currentPlayer: Player
landOnCenterSquare()
getColors()
list[Color]
selectCategory(color)
category
pickRandomQuestion()
question
question
evaluateAnswer(answer)
showAnswer()
String
alt
[answer is correct]
isWinner()
boolean
alt
[player is winner]
endGame()
map
[player is not winner]
isAnswerCorrect: boolean
startTurn()
changeTurn(nextPlayerIndex)
[answer is incorrect]
isAnswerCorrect: boolean
startTurn()


Use case 5: Manage Questions
QuestionStats
Question
Category
Actor
System
selectManageQuestions()
requestAuthentication()
enterCredentials(username, password)
displayQuestionManagerUI()
alt
[Teacher Action]
[Modify Existing Question]
selectQuestion(questionId)
getQuestionDetails()
questionDetails
displayQuestionModifyUI(questionDetails)
modifyQuestion(newDetails)
updateDetails(newDetails)
updateStats()
[Add New Question]
selectAddNewQuestion()
displayQuestionAddUI()
addNewQuestion(details)
createNewQuestion(details)
addQuestion(newQuestion)
initializeStats()
[Delete Question]
selectQuestion(questionId)
deleteQuestion(questionId)
removeQuestion(questionId)
selectReturnToHomeScreen()
displayHomeScreen()

Version 1.0 - 7/29/2024


Use case 6: Manage Categories
Category
Actor
System
selectManageQuestions()
requestAuthentication()
enterCredentials(username, password)
displayCategoryManagerUI()
alt
[Teacher Action]
[Modify Existing Question]
selectCategory(categoryId)
getCategoryDetails()
categoryDetails (DASHED)
displayCategoryModifyUI(categoryDetails)
modifyCategory(newDetails)
updateDetails(newDetails)
setName(newName)
[Add New Question]
selectAddNewCategory()
displayCategoryAddUI()
addNewCategory(details)
createNewCategory(details)
addCategoryToList(newCategory)
[Delete Question]
selectCategory(categoryId)
deleteCategory(categoryId)
removeCategoryFromList(categoryId)
selectReturnToHomeScreen()
displayHomeScreen()


Use case 7: Manage Account
Actor
System
selectManageAccounts()
requestAuthentication()
enterCredentials(username, password)
displayAccountManagerUI()
alt
[Teacher Action]
[Modify Existing Account]
selectAccount(accountId)
displayAccountModifyUI(accountDetails)
modifyAccount(newDetails)
confirmAccountModified()
[Add New Account]
selectAddNewAccount()
displayAccountAddUI()
addNewAccount(details)
confirmAccountAdded()
[Delete Account]
selectAccount(accountId)
deleteAccount(accountId)
confirmAccountDeleted()
selectReturnToHomeScreen()
displayHomeScreen()

Lessons Learned

We used ChatGPT AI to assist with translating our original class diagram image, which contained attributes and operations, into a starting outline for the Class Specifications in text format, as well as to help with proofreading and sentence enhancement. While it provides additional information and focuses on organizing the content, it is not without flaws and requires vigilant supervision and manual correction to meet our vision.

Credits List

Casey - Worked on the Static Class Diagram, created the Class Specifications, Created the Database Design Requirements

Calvin - Edits to Dynamic diagram. Minor edits to class specifications.

Ting-Wei Wang - Aid modify Static Class Diagram, create Dynamic Design

Justin - Worked on the Static Class Diagram, minor edits to Class Specifications, edit Dynamic diagram

Annex

Database Requirements

The database, implemented using Firebase Firestore, stores:

1. **Questions and Answers:**
    - Question text
    - Answer options (for multiple choice questions)
    - Correct answer(s)
    - Category
    - Difficulty level
    - Type (text, image, audio, video)
    - Usage statistics (times asked, correct answer percentage)
2. **User Profiles:**
    - User ID
    - Username
    - Email (for authentication)
    - Password hash (if using email/password authentication)
    - Game history (games played, win/loss record)
    - Achievement tracking
3. **Game Sessions:**
    - Session ID
    - Participating players
    - Current game state (board configuration, player positions, scores, question)
    - Game settings (number of questions, categories in play, custom rules)
    - Chat logs (if implementing a chat feature)
4. **Media Content:**
    - References to cloud storage locations for audio, video, and image content associated with questions
    - Metadata (file type, size, duration for audio/video)
5. **Categories:**
    - Category names
    - Descriptions
    - Associated color or icon references
    - Questions

Implementation

Firestore

Firebase Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud Platform. It is a NoSQL, document-oriented database that allows you to store, sync, and query data for your applications. Firestore organizes data into collections of documents, each containing key-value pairs, and supports real-time data synchronization, offline capabilities, and robust querying features. This makes it suitable for building dynamic, responsive applications that require a reliable and efficient data management solution.

Firestore houses data using key-value pairs like json:

- Categories
    - Geography (List of Questions)
        - Question Unique ID (Each Question will have the following)
            - Date Created
            - Difficulty level
            - Usage statistics (times asked, correct answer percentage)
            - Creator
            - Answer
            - Type of question [text, audio, image, video]
            - If text:
                - Question
                - textType [Open ended, multiple choice]
            - Else:
                - File Location (comes from firebase storage)
            -
    - Art_Literature (List of Question)
        - Question Unique ID
            - …
    - Science_Natrue (List of Question)
        - Question Unique ID
            - …
    - History (List of Question)
        - Question Unique ID
            - …
    - Sport_Leisure (List of Question)
        - Question Unique ID
            - …
- Users (list of users)
    - User Unique ID (Each users will have the following)
        - Name

        - email
        - Grade
        - Game history (games played, win/loss record)
        - Achievement tracking
- GameSessions
    - Timestamp
        - GameSessionID
        - Participating players
        - Current game state (board configuration, player positions, scores)
        - Game settings (number of questions, categories in play, custom rules)
        - Chat logs (if implementing a chat feature)
        - Question id list (list of questions that are attached to a game)

Firebase Object Storage

Cloud object storage is a data storage architecture that manages data as objects, rather than as files within a file system or as blocks within sectors on a disk. Each object includes the data itself, a variable amount of metadata, and a globally unique identifier. This approach provides scalable, cost-effective, and flexible storage that is ideal for storing large amounts of unstructured data such as documents, images, videos, and backups. Cloud object storage is accessed via APIs and can be managed through web interfaces provided by cloud service providers, allowing for easy integration with applications and services.

Caffeine Coders will utilize Firebase Storage as the cloud data storage solution for videos, audio, and images required for the questions.

All data will be stored in a bucket, which will have a unique identifier accessible via APIs. The current bucket ID is: `gs://caffeinecoders-e8219.appspot.com.`

Folders can be created within the bucket, functioning like directories. Caffeine coders will have a folder for the following modalities of objects:

1. /Videos,
2. /Audio
3. /Images

For example, a folder might be: `gs://caffeinecoders-e8219.appspot.com/videos.`

Data within each folder can be downloaded from the bucket using API calls that reference the file location. For instance, a file location could be:
`gs://caffeinecoders-e8219.appspot.com/images/caffeine_coders.jpeg.`

Once a file is uploaded, we can store its location string in Firestore, allowing our app to reference it. Essentially, we are storing a pointer to the object inside Firestore. This method enables us to categorize questions, track answers, and manage everything within a single database.